INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel. 954 90 20

Rapports de Recherche

N° 133

LE SYSTÈME PERLUETTE ET LES PREUVES DE REPRÉSENTATION DE TYPES ABSRAITS

3 Po Reçu le 9. 6 82.
Reste 320.

Eric MADELAINE

Mai 1982

LE SYSTEME PERLUETTE ET LES PREUVES DE REPRESENTATION

Eric MADELAINE

RESUME

Le système Perluette utilise une sémantique en termes de types abstraits algébriques des langages de programmation. Les spécifications algébriques permettent de faire la preuve de correction des compilateurs produits par le système d'une manière modulaire et en grande partie automatique.

Nous montrerons ici sur un exemple simple de types abstraits algébriques (tiré de [Gut 77]) comment peut se faire la preuve de correction de la représentation, à l'aide d'un démonstrateur de théorèmes : LCF. Nous décrirons les stratégies de preuve les plus utiles dans notre domaine et montrerons comment on peut détecter des erreurs dans la spécification des types ou de la représentation.

ABSTRACT

The PERLUETTE system uses a semantics of programming languages based on algebraic abstract data types. Algebraic specifications allow us to prove the correctness of the compilers produced by the system in a very modularized and automatic way.

We show on a very simple example of algebraic data type (from [Gut77]) how we can prove the correctness of its representation, using the LCF theorem prover. We detail our most usefull proof tactics and the way we detect errors in the type specification or in the representation function.

Mots clés :

Types abstraits algébriques
Métacompilation
Démonstrateurs automatiques.

RESUME

Le système Perluette utilise une sémantique en termes de types abstraits algébriques des langages de programmation. Les spécifications algébriques permettent de faire la preuve de correction des compilateurs produits par le système d'une manière modulaire et en grande partie automatique.

Nous montrerons ici sur un exemple simple de types abstraits algébriques (tiré de [Gut 77]) comment peut se faire la preuve de correction de la représentation, à l'aide d'un démonstrateur de théorèmes : LCF. Nous décrirons les stratégies de preuve les plus utiles dans notre domaine et montrerons comment on peut détecter des erreurs dans la spécification des types ou de la représentation.

1) Systèmes de preuve dans les types abstraits

Il existe à l'heure actuelle un certain nombre de systèmes permettant d'écrire des spécifications algébriques de types abstraits. Certains comprennent un démonstrateur de théorèmes qui permet de faire des preuves dans les théories obtenues.
AFFIRM [AFF 79] a certainement été un des premiers. Il permet de construire des spécifications algébriques hiérarchisées et de faire de la preuve de théorèmes, y compris par induction structurelle. Cependant, l'utilisateur doit guider la preuve en permanence, le système faisant peu de chose automatiquement. L'algorithme de Knuth et Bendix qui est incorporé a parfois des comportements surprenants.

OBJ [GF 79] , et CLEAR [BG 80], permettent de spécifier des types hiérarchisés et paramétrés. La paramétrisation rend les spécifications plus modulaires, mais la sémantique est nettement plus complexe [ADJ 80].
D'autre part, à part les travaux récents de Sanella [SW 82], ni OBJ ni CLEAR ne permettent de faire de la preuve de théorèmes.

La plupart des autres systèmes ne comportent pas de possibilités de preuves, sauf PAT [LW82] qui est implanté comme une extension de LCF.

Notre problème est plus particulier puisque nous nous intéressons uniquement aux spécifications algébriques des langages de programmation, et à leur preuves de représentation. Après quelques essais en PROLOG [BD 81], nous avons opté pour LCF, qui nous fourni des outils puissants pour programmer les tactiques de preuve spécifiques à notre domaine [Ma 81].

2) LCF

Le système LCF [LCF 77] comporte deux parties : un Meta langage: ML qui est un langage interactif fonctionnel fortement typé, et un système d'écriture de théories algébriques : PPLAMBDA (pour "Polymorphic Predicate Lambda Calculus").

PPLAMBDA est un langage permettant d'écrire des "termes", des formules" et des "types". Les termes cont ceux du lambda calcul typé. Les types sont des domaines de Scott, comprenant chacun un ordre partiel << (ordre d'information) et un plus petit élément UU (indéfini). Deux types sont prédéfinis, ainsi que quatre constructeurs de type :

tr	le type des valeurs de vérité
.	le type ne contenant qu'un seul élément celui-ci est dénoté ( ).
n	ajoute un élément UU à un type
x	le produit cartésien
+	la somme disjointe
→	le domaine des fonctions continues

Les formules sont des égalités entre termes (= =) ou des inégalités (<<) ou sont construites par conjonction (*&*), implication (*IMP*) et quantification universelle(!). Les théorèmes sont soit des axiomes, soit des formules prouvées, c'est à dire construites par les règles d'inférences.

Pour implanter des types abstraits algébriques, nous utiliserons la notion de construction (hiérarchique) de théories. On peut, en ML, construire une nouvelle théorie à l'aide des procédures suivantes :

newtype introduit un opérateur de type (pour nous une nouvelle sorte, paramètrée ou non).

newconstant introduit symbole de fonction et signature.

newparent importe le contenu d'une autre théorie.

newaxiom introduit un axiome (conditionnel positif).

maketheory fige une théorie, de sorte qu'on ne puisse lui ajouter ni nouveaux symboles, ni nouveaux axiomes.

Les preuves sont faites à l'aide des "tactiques" de ML : un but (*goal*) est un triplet contenant la formule que l'on veut prouver, un ensemble de règles de réécriture appelé "*simpset*", et une liste d'hypothèses. Une "*tactic*" est une fonction qui transforme un "*goal*" en une liste de "*goals*" (plus simples) et inversement, une "*proof*" est une fonction qui transforme une liste de théorèmes (déjà prouvés) en un théorème.

Pour prouver une propriété w , on décomposera donc le but (w, ss, hyp) en buts plus simples, jusqu'à obtenir une liste de sous-buts vides, puis on recomposera les fonctions de preuve selon le chemin inversé, pour construire le théorème w.

ML fournit un certain nombre de tactiques prédéfinies, ainsi que de "*tacticals*", qui servent à combiner les tactiques élémentaires.

3) Exemple : Représentation des tables de symboles

Il existe plusieurs moyens de spécifier la représentation d'un TAA par un autre. [GAU 80] : par une fonction d'abstraction, qui va des objets concrets vers les objets représentés (deux objets "concrets" peuvent représenter le même objet "abstrait"), ou par une fonction de représentation qui associe à chaque terme du type source un terme du type représentant. C'est cette dernière approche qui nous interessera, parce qu'elle est mieux adaptée aux problèmes de réécriture.

Dans son article [Gut 77] Guttag spécifie à la fois les deux approches mais plutôt que de construire l'union des types objet et source, il redéfini pour chaque fonction f du type source une fonction f' de profil correspondant dans le type objet. Nous utiliserons plutôt une fonction *rep*, de profil *Symboltable* → *Stack*, qui fera correspondre aux termes du type source des termes du type objet, au lieu d'associer directement les symboles de fonction des deux types.

C'est une approche équivalente, mais qui permet d'éviter d'introduire de nouveaux symboles de fonction dans le type objet.

De manière plus générale, pour représenter une structure source hiérarchisée par une structure objet (elle aussi hiérarchisée), nous utiliserons une fonction de représentation polymorphique. A tout terme d'une sorte donnée du type source, elle fera correspondre un terme de la sorte correspondante du type objet.

Il est un cas dans lequel nous ne pourrons pas éviter d'ajouter un symbole de fonction dans le type objet, c'est le cas des règles de représentation récursives. Dans notre exemple, la représentation de la fonction *Retrieve* est intrinséquement récursive, et nous devons introduire un symbole *Retrieve'* dans le type objet, qui représente directement *Retrieve* ; la définition (récursive) de *Retrieve'* ne présente alors plus de problème, puisqu'on travaille directement sur les termes objets.

Pour assurer la suffisante complétude de cette définition, on peut soit en prouver la terminaison, soit l'écrire sous la forme de trois règles agissant sur les constructeurs du type *stack*. Nous ne l'avons pas fait ici pour ne pas alourdir l'exemple.

Type : ST

Constructors :	Init	( ) → ST
	Enterblock :	(ST)→ ST
	Add :	(ST, ID, ALIST) → ST
Opérations :	Leaveblock :	(ST)→ ST
	Isinblock :	(ST,ID)→ Bool
	Retrieve :	(ST,ID)→ ALIST

Axioms :

```
Leaveblock  Init = Error
Leaveblock (Enterblock st) =  st
Leaveblock (Add(st,id,at)) = Leaveblock st
Isinblock  (Init,id) = False
Isinblock  (Enterblock st,id) = False
Isinblock  (Add (st,id,at)) =  If  id = id1
                               Then  True
                               Else Isinblock (st,id1)

Retrieve  (Init, id)  =  Error
Retrieve  (Enterblock st,id)  =  Retrieve (st,id)
Retrieve  (Add(st;id,at) , id1) =
                  If  id = id1
                  Then  at
                  Else  Retrieve (st,id1)
```

Figure 1

Le type source : ST

On remarquera que la spécification est suffisamment complète, ce qui nous permet de faire l'induction structurelle sur les trois constructeurs Init, Enterblock, et Add.

```
Types :  array, stack
Operations :
          empty :        ( ) → array
          assign :       (array,ID,ALIST) → array
          read :         (array,ID) → ALIST
          isundefined:   (array,ID) → Bool

          newstack :     ( ) → stack
          push :         (stack, array) → stack
          pop :          (stack) → stack
          top :          (stack) → array
          isnewstack :   (stack) → Bool
          replace :      (stack, array) → stack

Axioms :
          isnewstack :   (newstack) = True
          isnewstack (push(stk,a)) = False
          pop(newstack) = Error
          pop(push(stk,a)) = stk
          top(newstack) = Error
          top(push(sth,a)) = a
          replace(stk,a) = If isnewstack(stk)
                           Then Erreur
                           Else push(pop(stk),a)

          isundefined(empty,id) = True
          isundefined(assign(a,id,at), id1) =
                    If id=id1 Then False
                              Else isundefined(a,id1)

read(empty,id) = Error
read(assign(a,id,at),id1) =
            If id = id1
            Then at
            Else read(a,id1)
```

Figure 2

Le type objet : Les piles (de tableaux)

```
rep(Init) = push(newstack, empty)
rep(Enterblock st) = pusk(rep st, empty)
rep(Leaveblock st) = If isnewstack(pop(rep st))
                     Then error
                     Else pop(rep st)

rep(Add(st id,at)) = replace(rep sk, assign((top(rep st)),id,at))
rep(Isinblock(st,id)) = If isnewstack(rep st)
                     Then False
                     Else If isundefined (top(rep st),id)
                          If False
                          Then True

rep(Retrieve(st,id)) = Retrieve'(rep st,id)

        . . . . . .

Retrieve'(sk,id) =  If isnewstack(sk)
                    Then Error
                    Else If isundefined(top sk),id)
                    Then Retrieve'(pop sk, id)
                    Else read(top sk, id)
```

Figure 3

La représentation

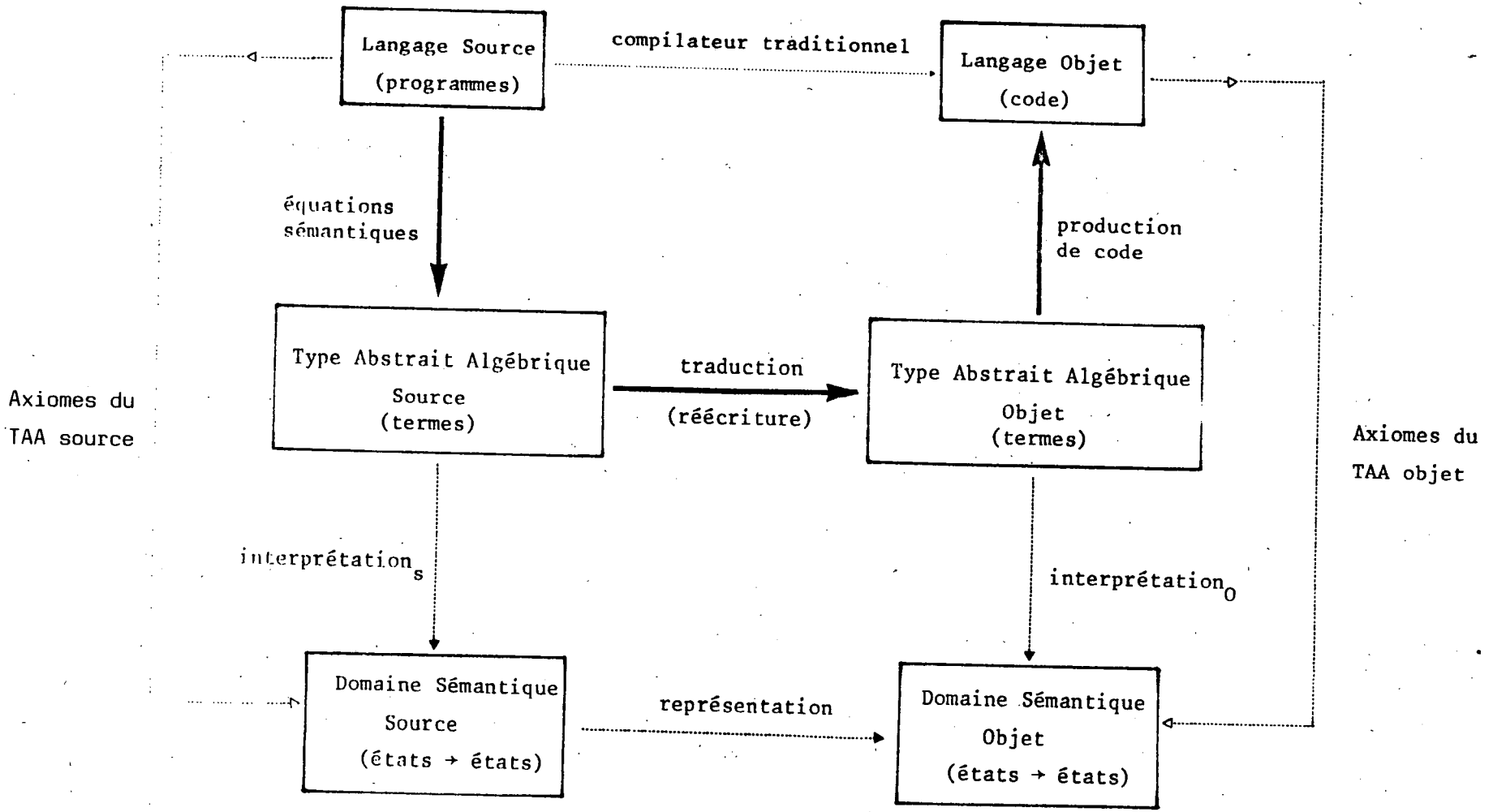
4) Perluette

1 - Que montrer ?

Pour fabriquer un compilateur, le système Perluette a besoin :

- d'une spécification d'un type abstrait source, accompagnée d'une grammaire attribuée assurant la traduction d'un programme source en un terme source,
- d'une spécification d'un type abstrait objet, accompagnée d'une description de la génération de code à partir du terme objet,
- d'une spécification de la représentation du type abstrait source par le type abstrait objet.

Si l'on parle uniquement de la construction du compilateur, Perluette n'a pas besoin de la partie axiomes des spécifications. Ce n'est qu'au moment de la preuve de correction que nous nous en servirons.
Qu'allons-nous prouver ? D'une part, la consistance et lorsque c'est possible, la suffisante complétude des types abstraits[Gau 80] et de la représentation. D'autre part, la correction de la représentation du type source par le type objet.



Principe de Perluette

Nous ne parlerons ici que de cette dernière partie. Précisons-là un peu : il s'agit de démontrer la validité des axiomes du type source à l'aide des axiomes du type objet d'une part, et des règles de représentation d'autre part. Plus précisément, pour tout axiome source de la forme :

"lhs = rhs"

il nous faut prouver la formule :

"rep(lhs) = rep(rhs)"

Dans certains cas, si nos deux types possèdent des sortes communes (Bool, Id ..), la représentation des objets de ces sortes sera réduite à l'identité. Prenons par exemple l'axiome (cf. figure 1) :

"Isinblock(Init,id) = Faux".

Il nous suffira de le traduire en :

"rep(Isinblock(Init,id)) = Faux".

2 - Construction de notre hiérarchie par Perluette

Les types abstraits que nous utilisons dans Perluette pour prouver la représentation d'un langage source par un langage objet sont spécifiés dans un langage de présentation de types, qui est le langage d'entrée du système. Trois des fichiers d'entrée nous intéressent ici : ils contiennent la spécification algébrique des types source et objet, et la fonction de représentation.

L'essentiel de la preuve - ou du moins la partie qui se fait en LCF - se déroule dans une théorie où l'on a mis :

- toute la spécification du type objet, sortes, signatures et axiomes,
- les sortes et signatures du type source, non compris les axiomes,
- la spécification de la fonction de représentation.

Les fichiers de construction de cette théorie, ainsi que le fichier contenant les formules à prouver (représentation des axiomes du type source) sont produits par Perluette. Il s'agit en fait de traduire nos spécifications de types abstraits en une syntaxe acceptable par LCF. A ce niveau, on peut également vérifier la consistance des spécifications et éventuellement la suffisante complétude des spécifications.

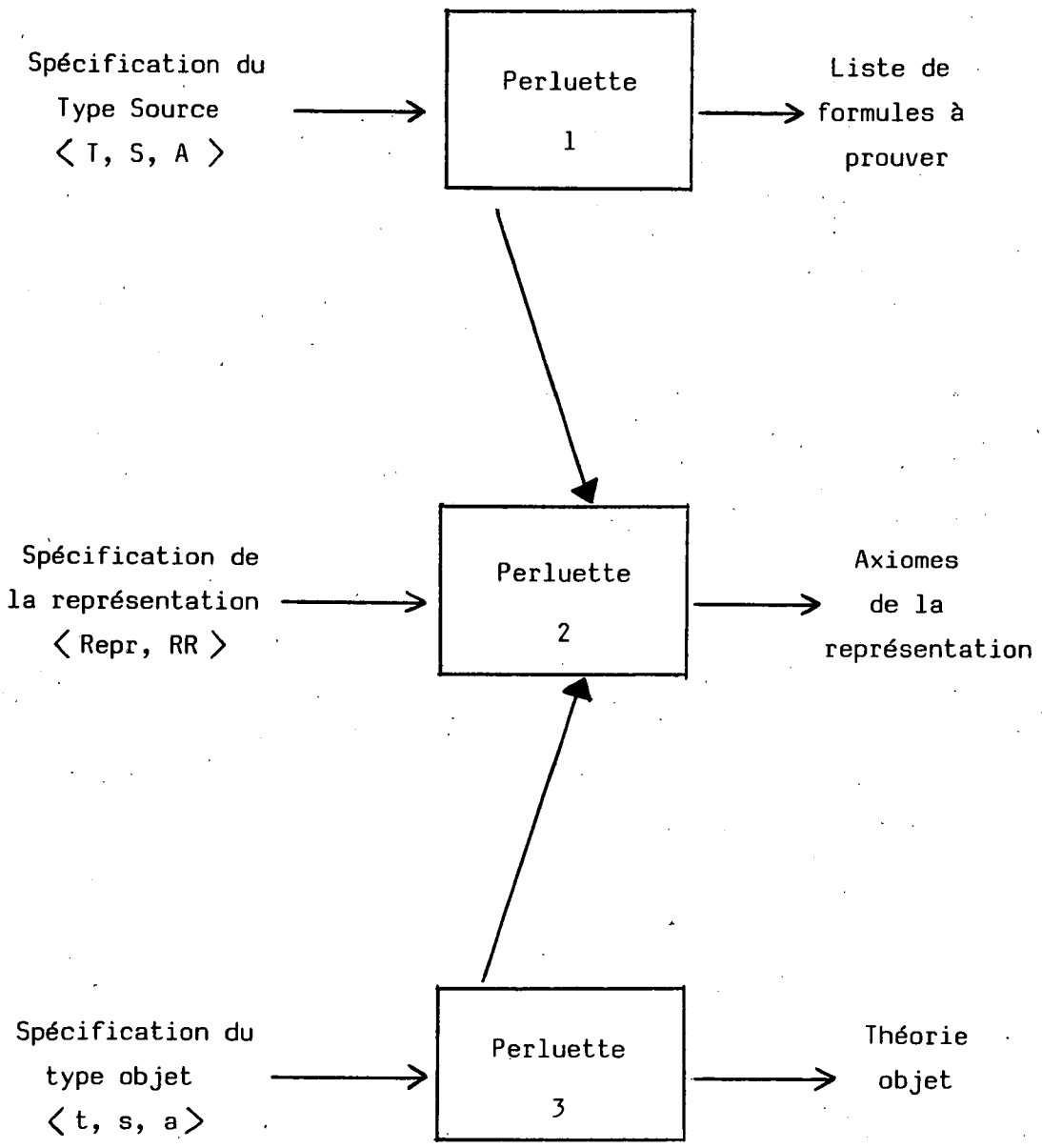


Figure 5

Schéma de production des fichiers destinés à LCF

5) La preuve

Nos fichiers de données ayant été traduits et entrés dans LCF : il nous reste le plus gros du travail : la preuve elle-même. Dans un premier temps, nous procéderons de manière très interactive, laissant à l'utilisateur le soin de deviner quelle tactique de preuve peut résoudre tel ou tel problème particulier. Dans la suite de ce chapitre, nous montrerons un par un les problèmes que nous ont posés les tables de symboles, et comment nous y avons répondu.

5.1) Réécriture

Quelle que soit la formule à démontrer, il y a une tactique que l'on utilisera à coup sûr à un moment ou à un autre, et souvent plusieurs fois lors de la preuve : la réécriture. C'est une tactique standard de LCF : *SIMPTAC*, qui "simplifie" la formule proposée en utilisant autant de fois que possible, dans un ordre non spécifié, tous les théorèmes figurant dans le *simpset*. On a vu que le *simpset* était une composante du but fourni à toute tactique ; on pourra ainsi spécifier par quel ensemble de règles on veut simplifier notre formule.

LCF ne fait bien sûr aucune vérification sur la nature des simpsets, et c'est à l'utilisateur de s'assurer que les ensembles de réécritures sont confluents, et à terminaison finie.

Considérons le deuxième axiome de *ST*. La formule qu'il nous faut montrer est :

w ="rep(Leaveblock (Enterblock st)) = rep st"

Pour ce faire, nous utilisons un ensemble de règles de réécriture (calculé par Perluette) qui contient :

- les règles définissant toutes les fonctions rencontrées dans les règles précédentes, et ainsi de suite, par fermeture transitive. (Pour notre exemple, les quatre règles définissant *isnewstack* et *pop*),
- les connaissance nécessaires au calcul des prédicats et relatives aux objets prédéfinis de LCF, contenues dans un *simpset* standard : *BASICSS*.

Nommons ce simpset *s1*, et proposons au système d'évaluer :

```
SIMPTAC(w,s1,[ ]) ;;
```

Réponse :

```
If isnewstack(rep st)
then UU
else(rep st) == rep st
```

Notre *simpset* est manifestement insuffisant. Mieux : le terme qui apparaît dans la conditionnelle, à la réflexion, est un invariant de représentation. Plus précisément, nous pouvons montrer (au paragraphe suivant), que la formule suivante est vraie pour tout terme du type *ST* :

```
!s:ST.   isnewstack(rep st) == FF
```

Après avoir prouvé ce théorème, ajoutons-le à l'ensemble de réécriture, et nommons *is1* le nouveau *simpset* cette fois-ci, la simplification rend une liste de sous-buts vide, et une fonction de preuve qui nous permet de construire le théorème souhaité.

5.2) Introduction Structurelle

L'induction structurelle est une méthode de preuve classique dans les types abstraits algébriques [Gau80]. Pour pouvoir l'utiliser en LCF, il va nous falloir l'adapter. Nous disposons dans la logique de Scott d'une seule règle d'inférence d'induction, qui est une induction de point fixe. Hélas, cette règle ne nous permet pas de valider directement l'induction structurelle sur un domaine plat, et nous devons introduire un axiome de bien-fondé de l'induction structurelle qui limite en fait la classe des modèles considérés :

Pour un type abstrait T, il nous faut supposer l'axiome suivant (et cette définition est généralisable aux types paramétrés) :

$\forall f:T\to T$	A:Axiome de bien-fondé
Si pour tout constructeur c de profil $T_1 x \ldots T_n \to T$	
$f(c(\vec{t}))=c(f(\vec{t}))$	
où $\vec{t}=(t_1,\ldots,t_n)$	
et $f(\vec{t})=(\hat{f}(t_1),\ldots\hat{f}(t_n))$ avec $\hat{f}(t_i)=\begin{cases} f(t_i) \text{ si } T_i=T \\ t_i \text{ sinon} \end{cases}$	
Alors, on a $\forall t:T,\ f(t)=t$	

Soit Tfun la fonctionnelle définie par :

∀f:T → T, ∀C constructeur de T,

Tfun f C(E) = C(f(E))

Alors l'axiome de bienfondé nous permet de prouver le théorème suivant :

	B: Théorème de point fixe
∀t: T, FIX Tfun t = t	

Pour pouvoir faire de l'induction structurelle sur un type, il faut donc introduire, dès la construction de la théorie correspondante (quand la théorie est "gelée", c'est trop tard !), les axiomes de définition de Tfun, et l'axiome de bien fondé; en pratique, nous ne referons pas chaque fois la démonstration du théorème de point fixe, que nous introduirons directement comme axiome, à la place de l'axiome de bien-fondé. (C'est une hypothèse en fait plus faible que l'axiome de bien-fondé, mais qui suffit pour utiliser l'induction de point fixe à la Scott). Une fois ceci fait, nous disposons d'une procédure *STRUCTAC* qui nous fabrique une tactique d'induction *STINDUC* relative au type ST. Cette dernière utilise à la fois l'induction de point fixe à la Scott et l'axiome B .

Sur notre exemple, l'application de *STINDUC* à notre invariant de représentation "isnewstack(rep st) = FF" nous donne les quatre sous-buts :

```
isnewstack(rep UU : ST) == FF
isnewstack(rep Init) == FF
isnewstack(rep st) == FF
        => isnewstack(rep(Enterblack st)) == FF
isnewstack(rep st) == FF
        => isnewstack(rep(Add st id at)) == FF
```

Les trois dernières formules se prouvent simplement par réécriture, mais la première est plus difficile. Le problème est qu'en interprétant nos types comme des domaines de Scott, nous leur avons ajouté un élément : UU . Deux solutions sont utilisables : on peut définir pour chaque type un prédicat d'appartenance, faux pour UU, et vrai sur tous les constructeurs. Ensuite chaque théorème devra être précédé d'une hypothèse d'appartenance pour toutes les variables quantifiées de notre type d'intérêt.

Nous préférons la deuxième, qui consiste à considérer que toutes nos fonctions sont strictes, sauf les prédicats qui rendent *Faux* au point UU. Ceci du moins pour les prédicats définissant des classes de termes du type abstrait, et ce sont en fait les prédicats utiles dans notre domaine. En pratique, pour toute formule à montrer par induction, nous introduisons d'ailleurs directement la propriété souhaitée pour UU sous la forme d'un axiome du type, ce qui est beaucoup moins lourd.

5.2) Disjonction de cas

C'est de l'induction structurelle sur le domaine des valeurs de vérité ! Bien sûr, LCF étant très savant sur le calcul des propositions et des prédicats, la disjonction de cas fait partie des tactiques standard du système. Sous deux formes : on peut soit préciser sur quel terme (du type prédéfini *tr*) on désire faire l'analyse par cas (*CASESTAC*), ou on peut laisser le système chercher de lui-même un terme sur lequel il appliquera la tactique précédente (*CONDCASESTAC*). Quelle que soit la tactique choisie, elle engendre trois sous-buts pour les trois valeurs possibles du terme logique, en ajoutant chaque fois dans le simpset et dans la liste d'hypothèses un théorème de la forme *terme* == UU (respectivement *FF*, *TT*).

```
rep(isinblock(add st id at, id1))
    == if(id=id1)
       then TT
       else(rep(isinblock(st, id1)))
```

```
       SIMPTAC(is2)

if if(id=id1)
   then FF
   else isundefined(top(rep st)) id1
then FF
else TT
      == if(id=id1)
         then TT
         else if is undefined(top(rep st)) id1
              then FF
              else TT
```

```
      CASESTAC "id=id1)

      THEN SIMPTAC(is2)
    TRUTH
```

Figure 6

Disjonction de cas

5.4) Règles récursives

On a vu au chapitre 3 que pour écrire une règle de représentation récursive, on est amené à introduire un nouveau symbole de fonction f' dans le type objet, et d'écrire une règle de la forme :

$$f'(x_1 \ldots x_n) = h\ (x_1 \ldots\ x_n)$$

où la partie droite $h(x_1 \ldots x_n)$ comporte au moins une occurrence de f'. Cette règle n'a pas la propriété de terminaison finie, et nous ne la mettrons pas dans un système de réécriture. Il n'existait pas de tactique standard pour appliquer une règle une fois et une seule (mais ça existait dans AFFIRM). Cependant, elle n'est pas très difficile à programmer en ML : nous avons deux procédures APTAC et APOCCTAC, qui prennent en argument un théorème, et qui rendent une tactique. Celle-ci applique le théorème (considéré comme une règle de réécriture) au premier sous-terme convenable de la formule - au $n^{ième}$ pour APOCCTAC -.

Sur notre exemple :

rep(Retrieve(Init, id)) = UU

Se réécrit par simplification en

Retrieve'(push neuwstack empty, id) == UU

Puis, utilisant une fois la définition de Retrieve'(règle *rep7*) par *APTAC rep7* :

```
If isnewstack(push newstack empty)
Then UU
Else If isundefined(top(push newstack empty),id)
Then Retrieve'(pop(push newstack empty),id)
Else read(top(push newstack empty))
                                == UU
```

Ce qui se réécrit immédiatement par *SIMPTAC* en :

UU == UU

5.5) Détection d'erreurs

Lors de la première tentative de preuve des neuf théorèmes du type *ST*, tout s'est très bien déroulé jusqu'à la huitième formule. Bien sûr, il faut un certain temps d'adaptation au système, dû, entre autres, au fait que les réponses de ML sont peu lisibles, mais il est rapidement facile de décider, à la vue d'un sous-but, avec quelle tactique il faut l'attaquer.

La huitième formule a donné, après application de APTAC et simplification, le sous-but curieux :

UU == Retrieve'(rep st, id)

Le fait que le membre gauche d'une formule qui était sensée être intéressante, puisse se réduire en UU alors que le membre droit est manifestement défini, est parfaitement déraisonnable. Ce qui laissait supposer une erreur quelque part dans la spécification de *Retrieve*. En effet, une lecture attentive du fichier représentation nous montrait rapidement une erreur dans la définition de *Retrieve'* (en fait une interversion des branches d'une conditionnelle).

La correction nécessitait bien sûr de reconstruire toute la théorie, et de reprendre les preuves depuis le début. Cette deuxième itération ayant permis de détecter une autre erreur, cette fois-ci dans la spécification de *replace*, ce n'est qu'à la deuxième reconstruction que les derniers théorèmes ont été prouvés.

6. Conclusion

Une des questions qu'il était légitime de se poser en abordant l'automatisation de la preuve est : comment le système pourra-t-il détecter, et localiser des erreurs dans la spécification ? Il apparaît que c'est certainement l'utilisateur lui-même qui devra intervenir directement. Mais, ceci après que LCF ait fait toute la partie simple du travail (simplification, induction et autres tactiques standards ...). L'expérience montre que l'on met ainsi en évidence des erreurs qui sont très difficiles à détecter "à la main", ne serait-ce que des fautes de frappe ! les erreurs plus fondamentales, portant par exemple sur les choix de représentation, se traduisent à la console par une impossibilité de prouver une propriété, ou même par une contradiction. La force de LCF réside ici dans sa robustesse : même un utilisateur maladroit ou malveillant ne pourra jamais prouver un théorème faux (si la théorie dans laquelle il travaille est consistante). C'est une particularité qui n'était pas vraie dans AFFIRM, où l'algorithme de Knuth-Bendix se permettait de temps en temps de modifier la théorie dans laquelle on travaillait, ni à plus forte raison en PROLOG, où la validité des preuves dépendait des capacités du programmeur.

Deuxième point : jusqu'où peut-on automatiser ? La production des fichiers d'entrée étant à la charge de Perluette, on peut dans le même temps produire non seulement les *simpsets* utiles, mais aussi prévoir de quelle tactiques particulières on peut avoir besoin pour certaines formules (*APTAC, STRUCTAC, APOCCTAC* ...) et les produire automatiquement. De même, on peut essayer a priori

de produire des tactiques ayant de fortes chances de réussite, au vu de la forme (syntaxique) et même de certains renseignements sémantiques (types ...) des formules à prouver. L'utilisateur n'aura donc à intervenir que dans les cas critiques.

Reste à expérimenter sur un cas réel, pour déterminer par exemple jusqu'à quel point il faut spécialiser nos ensembles de réécriture, et comment la structure hiérarchique des types permettra de modulariser la preuve.

Bibliographie :

[ADJ80] EHRIG, KREOWSKI, THATCHER, WAGNER & WRIGHT "Parameterized Data Types in Algebraïc Specified Languages (Short version)", 7th ICAALP, juillet 80, LNCS n° 85.

[Aff79] D.H. Thompson : "AFFIRM reference manual"
S.L. GERHART : "AFFIRM type library"
S. LEE and S.L. GERHART : "AFFIRM user's guide"
USC/ISI Marinadel Bey. California.

[BD81] M. BERGMAN, P. DERANSART "Abstract Data types and rewriting systems : application to the programming of algebraïc abstract data types in PROLOG". 6th CAAP Genova. Italy 81, LNCS n° 112.

[BG80] R.M. BURSTALL, J.A. GOGUEN "Semantics of CLEAR"LNCS n° 86, 1980.

[Gau80] M.C. GAUDEL, "Génération et Preuve de compilateurs basées sur une sémantique formelle des langages de programmation" thèse d'état, INPL, mars 80.

[GT79] J.A. GOGUEN, J. TARDO "An introduction to OBJ-T" Specif. of reliable software, IEEE, Cambridge Mass., April 79.

[Gut77] J. GUTTAG "Abstract Data Types and the development of data structures", CACM, June 77, Vol 20-G.

[LCF77] M.GORDON, R. MILNEN, C. WADSWORTH "Edinburg LCF" LNCS n° 78, 1977.

[LW82] J. LESZCZYLOWSKI, M. WIRSING "A system for reasoning within and about algebraïc specification", unpublished draft.

[Ma81] E. MADELAINE "Outils de preuve sur les types abstraits algébriques" Rapport de DEA, Univ. Paris VII, Juin 81.

[SW82] D. SANELLA, M. WIRSING "Implementations of Parameterized algebraïc specifications", 9th ICALP, Aarhus 82.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique